Massachusetts Institute of Technology
Department of Electrical Engineering and Computer Science

6.826 – Principles of Computer Systems

Handout 15 September 27, 1993

Problem Set #3

Due date: October 4, 1993

Problem 1.

In this question you are to give a precise definition of the semantics for an alternative choice operator on commands. The "or" operator on commands is sometimes called "demonic choice", because if either of the component commands loops indefinitely, the composite command can also loop.

So, for example, the command "`DO SKIP OD [] x:=3`" can loop. (It can also set `x` to `3` and terminate.) An alternative operator, which we will denote by the symbol "`<>`", is sometimes called "angelic choice" because it is guaranteed to terminate if either of its component commands terminates. Thus, for example, the command "`DO SKIP OD <> x:=3`" has only one possible outcome, which involves setting `x` to `3` and terminating.

In the style of Handout 4, give a precise definition of the semantics of `<>`.

Problem 2.

Consider the following specification of a storage manager:

```
MODULE StrgMgr [T] =

TYPE S = SEQ[T]              % The allocated storage.
     N = Int                 % Names for allocated entities.
     Freed = SET[N]          % Set of freed names.

VAR s := S{}
    freed := Freed{}

APROC Malloc (t) -> N =      % Allocate and initialize a new T.
  << VAR n := s.size |
     s := s ++ t;
     RET n >>
```

```
APROC Free (n) =            % Free the storage named by n.
  << freed := freed ++ n >>

APROC Write (n,t) =         % Write a new value at location n.
  << n IN freed \/ n < 0 \/ n>=s.size => HAVOC
    [*] s[n] := t >>

APROC Read (n) -> T =       % Read the value stored at location n.
  << n IN freed \/ n < 0 \/ n>=s.size => HAVOC
    [*] RET s[n] >>

END StrgMgr
```

Now consider the following implementation, which attempts to minimize the amount of storage in use:

```
MODULE StrgMgrImpl [T] =

TYPE S = SEQ[T]             % The allocated storage.
     N = Int                % Names for allocated entities.
     Free = SET[N]          % Indexes of s available for reuse.
     OMap = N -> N          % Map from external to internal names.

VAR s := S{}
    omap := OMap{}
    free := Free{}
    nextN: N := 0           % Next external name to allocate.

APROC Malloc (t) -> N =     % Allocate and initialize a new T.
  << free.size = 0 =>
            s := s ++ t;
            omap(nextN) := s.size-1;
            nextN := nextN+1;
            RET nextN-1
   [*] VAR n | n IN free =>
            free := free - n;
            s[n] := t;
            omap(nextN) := n;
            nextN := nextN+1;
            RET nextN >>

APROC Free (n) =            % Free the storage named by n.
  << free := free ++ omap(n) >>
```

```
APROC Write (n,t) =          % Write a new value at location n.
  << s[omap(n)] := t >>

APROC Read (n) -> T =        % Read the value stored at location n.
  << RET s[omap(n)] >>

END StrgMgrImpl
```

Use history variables to define an abstraction function for the implementation, and sketch a proof of correctness using your abstraction function and any invariants that you need. Note: your abstraction function will have to define a mapping on `N`'s (i.e., the names used for allocated entities), since the implementation recycles them.

Problem 3.

In the previous question, we would have liked to implement the storage manager without using the omap. What assumptions would we need to make about the clients of the storage manager module for such an implementation to be correct?

Problem 4.

Recall the standard Unix representation of one file as it was explained in class.

(a) Write in Spec the state for this representation; decide any minor details that you don't remember or that weren't given in class any way you like. Give the abstraction function from this representation to the specification of the state of one file given in Handout 9.

(b) Write in Spec the state for the log used in the log structured file system as it was explained in class, assuming that the standard Unix representation of (a) is used as the index structure. For this purpose treat the log as a `SEQ Byte` rather than as a collection of disk pages. Give the abstraction function from the representation of a single file in your log to the standard Unix representation.

(c) Add to the state you give in (b) the volatile state needed for a practical implementation as described in class (that is, a cache of all the index blocks whose abstract state differs from the state recorded on the disk). Give the invariant satisfied by the combined state. Write in Spec a description of the algorithm for reconstructing the volatile state from the log after a crash. You don't have to worry about the representation of Free, just about the indexing information.